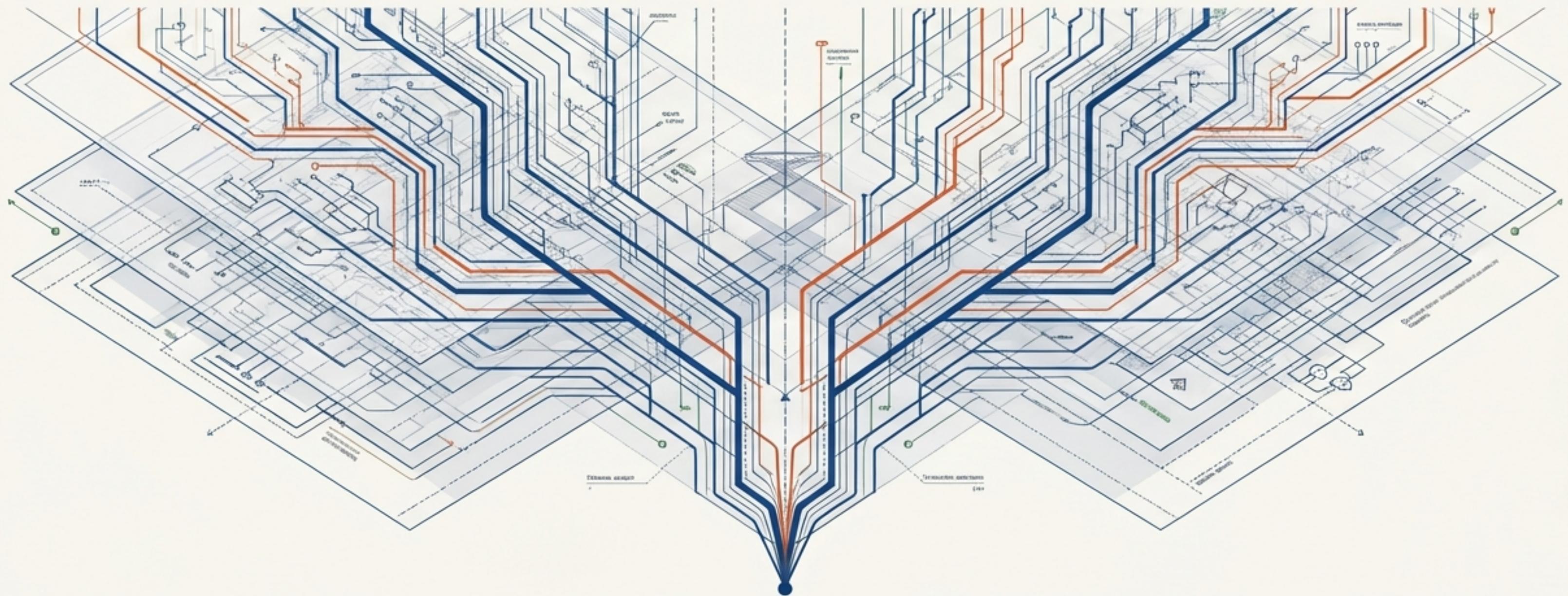


J-code



Разработка языка программирования для игр с
адаптивным, интеллектуальным противником.

За пределами детерминированных головоломок

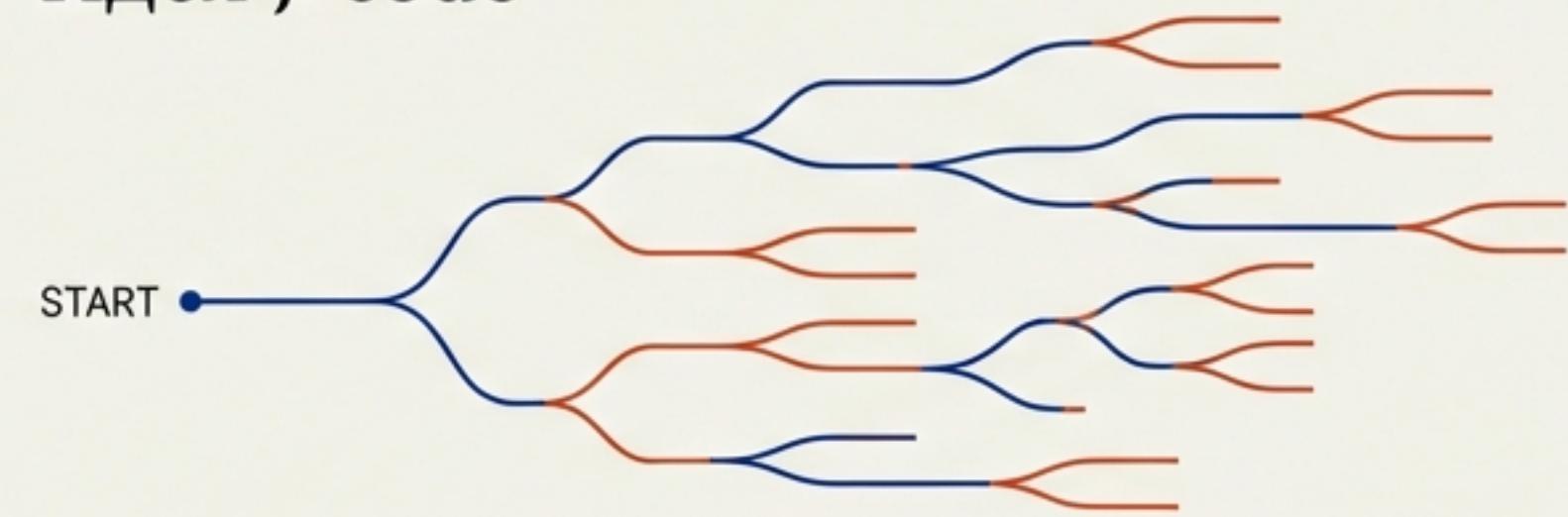
Большинство игр-головоломок имеют оптимальное решение. Что, если игра будет активно мешать вам его найти?

Стандартные игры



- Детерминированы. Для одинаковой последовательности ходов результат всегда один и тот же.
- Примеры, как в некоторых версиях Тетриса, всегда подбрасывают ту фигуру, которая подходит меньше всего, но делают это по предсказуемому алгоритму.

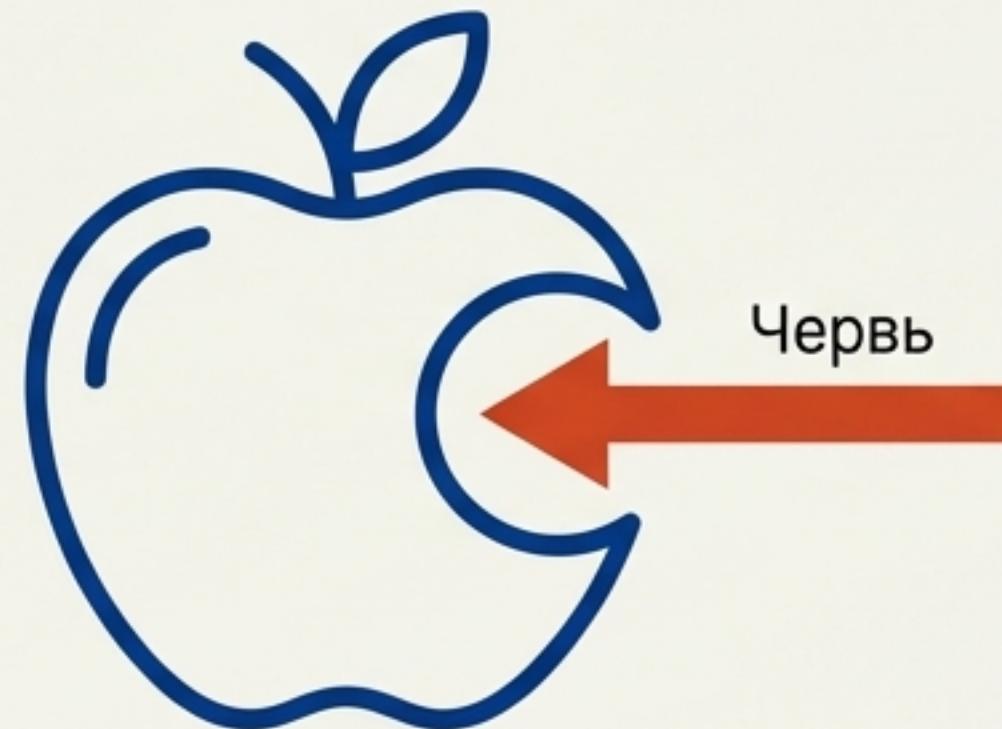
Идея J-code



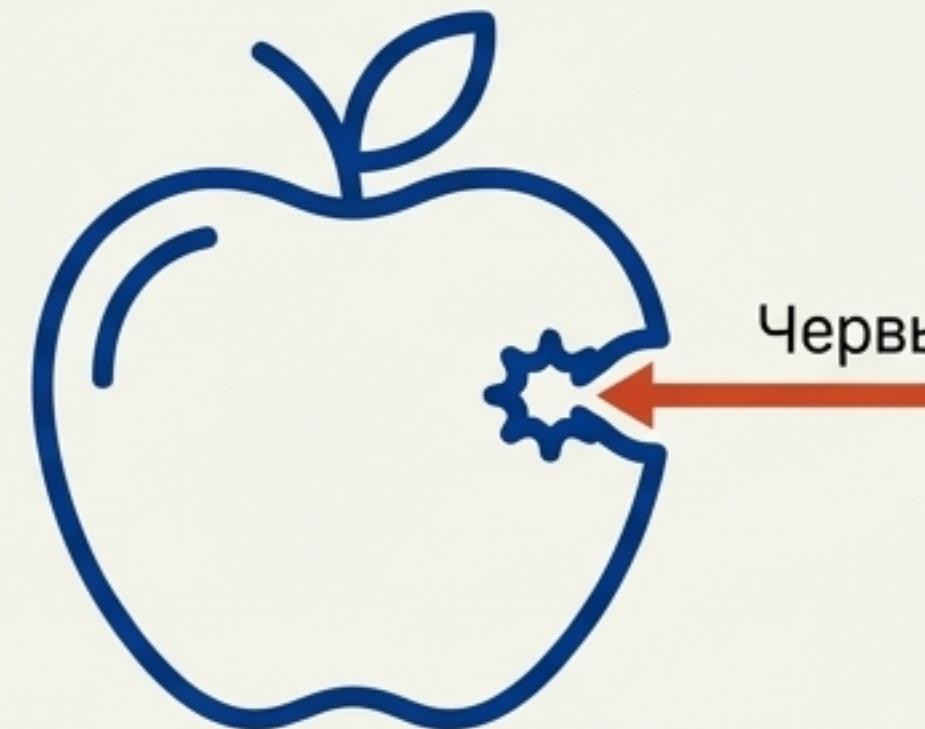
- Создать непредсказуемого противника. Игра анализирует вашу стратегию и находит в ней «дыры», чтобы использовать их против вас.
- Цель: Не просто решить задачу, а написать программу, способную противостоять «умному» противнику, который адаптируется к вашим действиям.

Концепция «Червь в яблоке»

Чем лучше ваш код, тем экспоненциально сложнее становится игра.



Слабая стратегия: быстрая игра,
быстрый проигрыш



Сильная стратегия: долгая игра,
сложный вызов

- Ваша программа — это «яблоко».
- Уязвимости и слабые места в вашей стратегии — это «дыры».
- Игра («червь») находит самую большую «дыру» и использует её, чтобы победить.
- **Нелинейная зависимость:** Небольшое улучшение кода (уменьшение «дыры») приводит к экспоненциальному росту продолжительности и сложности игры. Червю становится труднее «залезть» и он «ест яблоко» медленнее.

Экосистема J-code: Иерархия из четырех ролей

Гибкость и мощность системы достигается за счёт четкого разделения ответственности.



- 0. Автор Языка (Я):** Реализация ядра интерпретатора на Python.
- 1. Системный Программист:** Создание базовых функций и библиотек на самом J-code. Вынос логики из Python для максимальной гибкости.
- 2. Автор Игры:** Проектирование правил и механик конкретной игры (например, «Змейка» или «Тетрис»).
- 3. Автор Уровня:** Определение начальных условий, конфигураций и состояний объектов для конкретного уровня.
- 4. Игрок:** Написание программы-решения, которая управляет игровыми объектами в ответ на действия игры.

Как игра находит ваш самый слабый ход?

Игра использует вашу программу как оракула для поиска худшего для вас сценария.

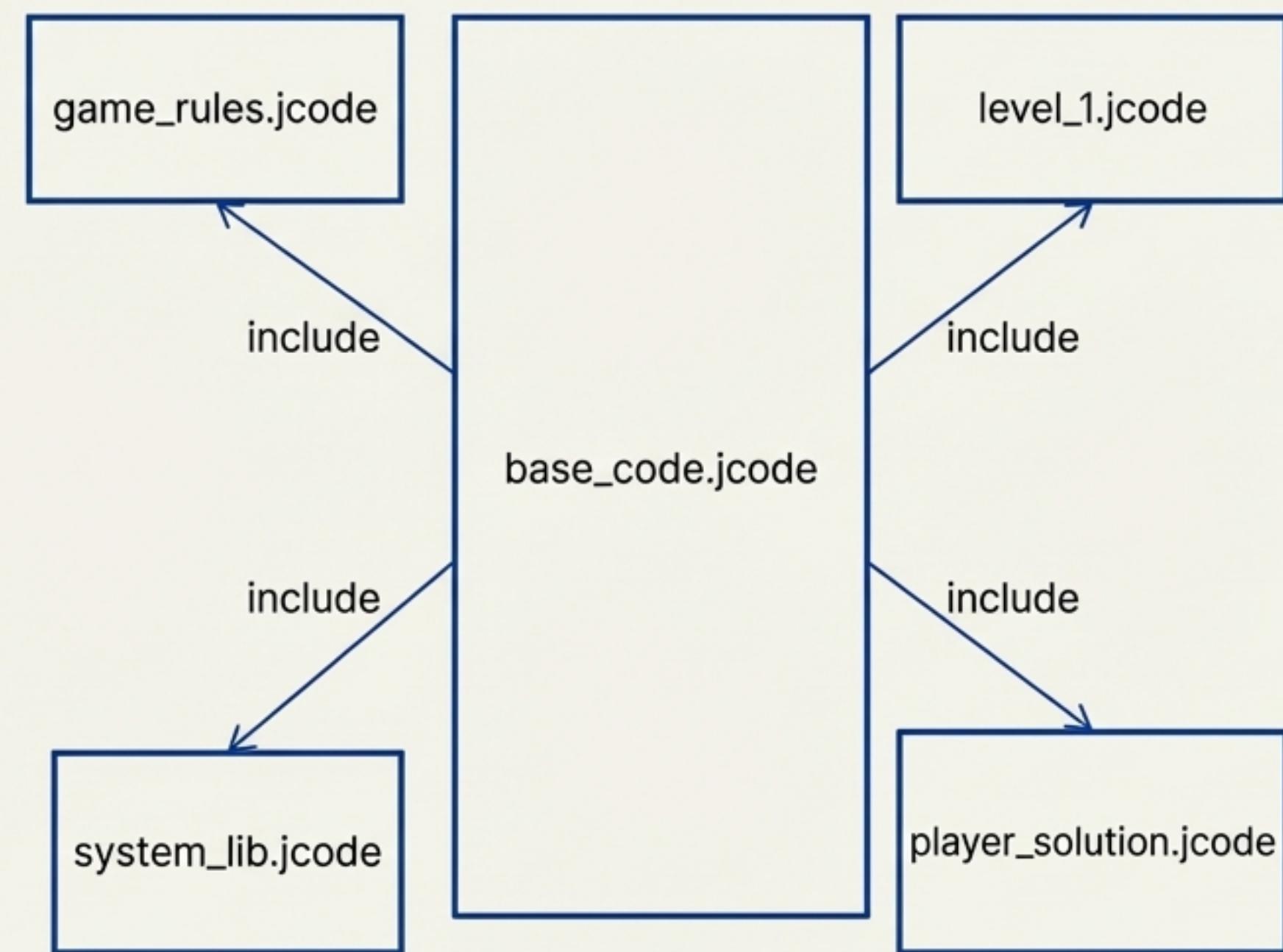
- Игра не пытается «угадать» вашу уязвимость. Вместо этого она моделирует будущее.
- Для каждого возможного следующего хода (например, для каждой из 7 фигур Тетриса) игра запускает вашу программу и смотрит, какую выгоду вы получите.
- Она выбирает и делает тот ход, который принесет вашей программе **минимальную выгоду**.
- **Результат:** Игра автоматически адаптируется к любой стратегии. Автору игры не нужно писать сложный ИИ — система находит уязвимости сама.



Модульная структура: Сила в файлах

Вместо монолитного кода — гибкая система из множества подключаемых файлов.

- Все компоненты системы — это отдельные .jcode файлы: системные функции, описание игры, уровни, решение игрока.
- Центральный механизм сборки — команда `include`.
- `include` позволяет рекурсивно и вложенно подключать файлы, создавая единое пространство выполнения.
Можно, например, подключить файл, который содержит список других файлов для подключения.



Синтаксис J-code: Простота и структура JSON

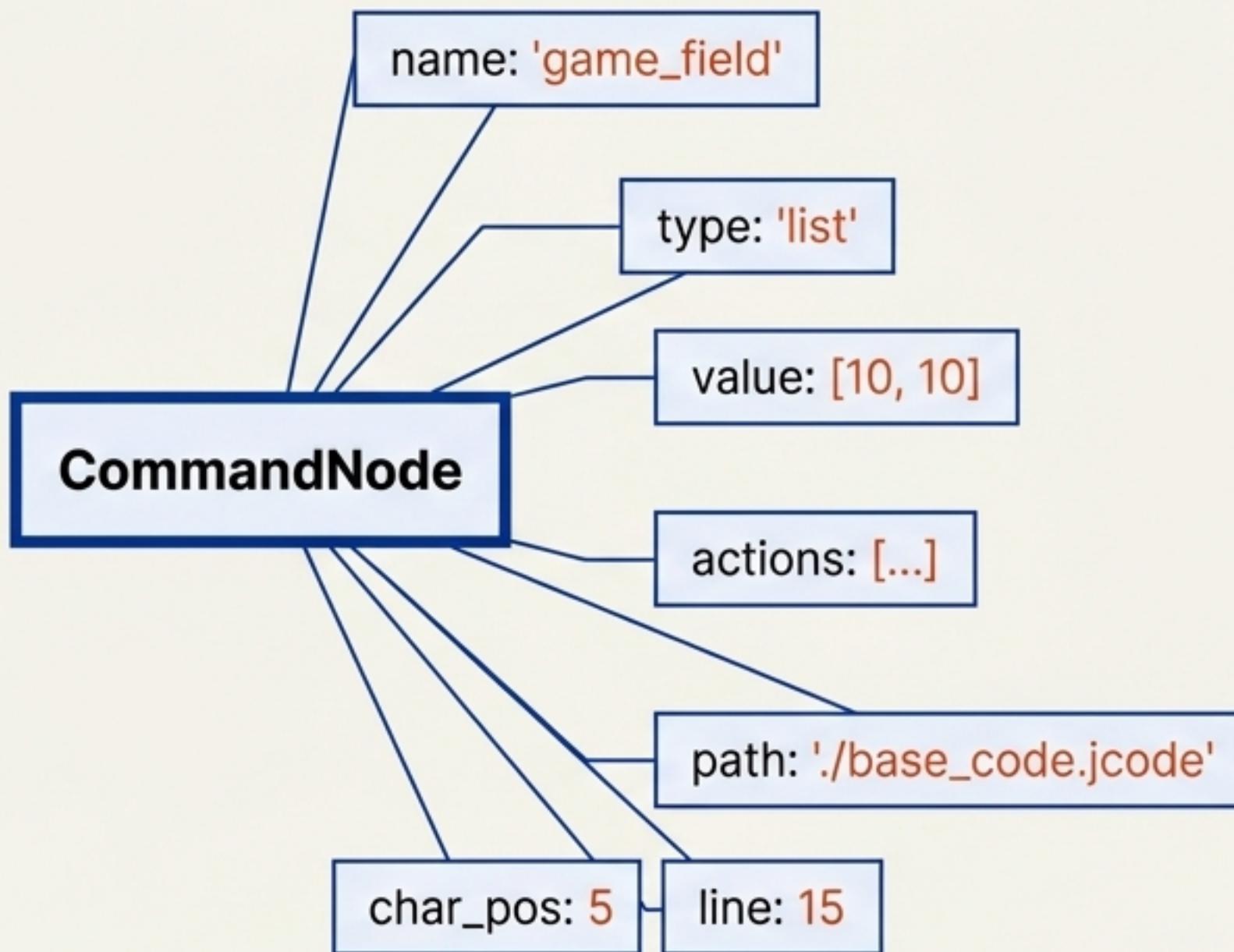
Код J-code — это валидный JSON, что обеспечивает строгую структуру и упрощает обработку.

- Основа языка — словари (`{}`) и списки (`[]`).
- Каждая команда — это объект JSON, где ключ — это имя команды, а значение — её тело (`"command_name": { ...body... }`).
- Тело команды строго типизировано (список, число, строка, словарь). Например, команда 'for' ожидает словарь с ключами 'in' и 'actions'.

```
{  
    "game_field": [10, 10],  
    "for": {  
        "in": 3,  
        "actions": [  
            { "move_forward": 1 }  
        ]  
    },  
    "include": "games/snake/player_solution.jcode"  
}
```

Под капотом: Единая сущность `CommandNode`

Все команды в коде, от `include` до `for`, являются экземплярами одного базового класса, что унифицирует их обработку.



- Класс `CommandNode` — это атомная единица языка в памяти.
- Он хранит не только суть команды, но и её точное местоположение в исходном коде для максимально детальных сообщений об ошибках.

Ключевые атрибуты:

- `name`: Имя команды (e.g., "game_field")
- `type`: Ожидаемый тип тела (e.g., "list")
- `value`: Тело команды
- `actions`: Вложенные команды
- `path`: Путь к файлу, где находится команда
- `line`: Номер строки
- `char_pos`: Позиция символа в строке

Сердце системы: Линковщик (Linker)

Линковщик — это механизм, который собирает, проверяет и готовит код к выполнению.

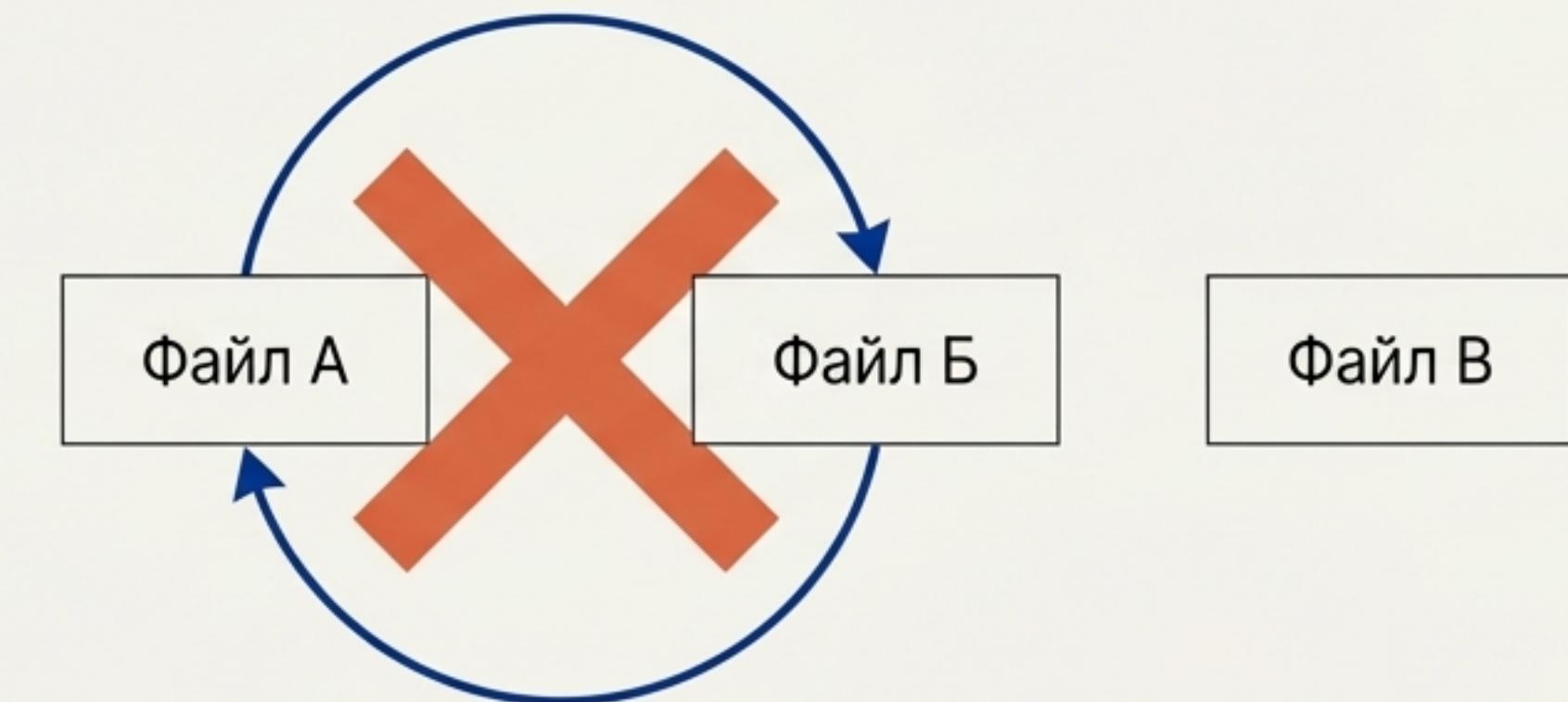
Перед тем как интерпретатор сможет выполнить код, линковщик выполняет несколько ключевых задач, превращая набор файлов в единое дерево команд.



Задачи Линковщика, Шаг 1: Построение и валидация деревьев

Первичная обработка файлов: проверка синтаксиса, построение структур и защита от бесконечных включений.

- Проверка синтаксиса JSON**: Убедиться, что все файлы корректно отформатированы (скобки, запятые).
- Построение дерева команд**: Преобразование кода в иерархическую структуру объектов `CommandNode`.
- Построение дерева `include`'ов**: Рекурсивный обход всех подключений для создания карты зависимостей файлов.
- Проверка на циклы**: Алгоритм обхода графа зависимостей для обнаружения циклов ($A \rightarrow A$; $A \rightarrow B \rightarrow A$; $A \rightarrow B$ и $B \rightarrow A$).



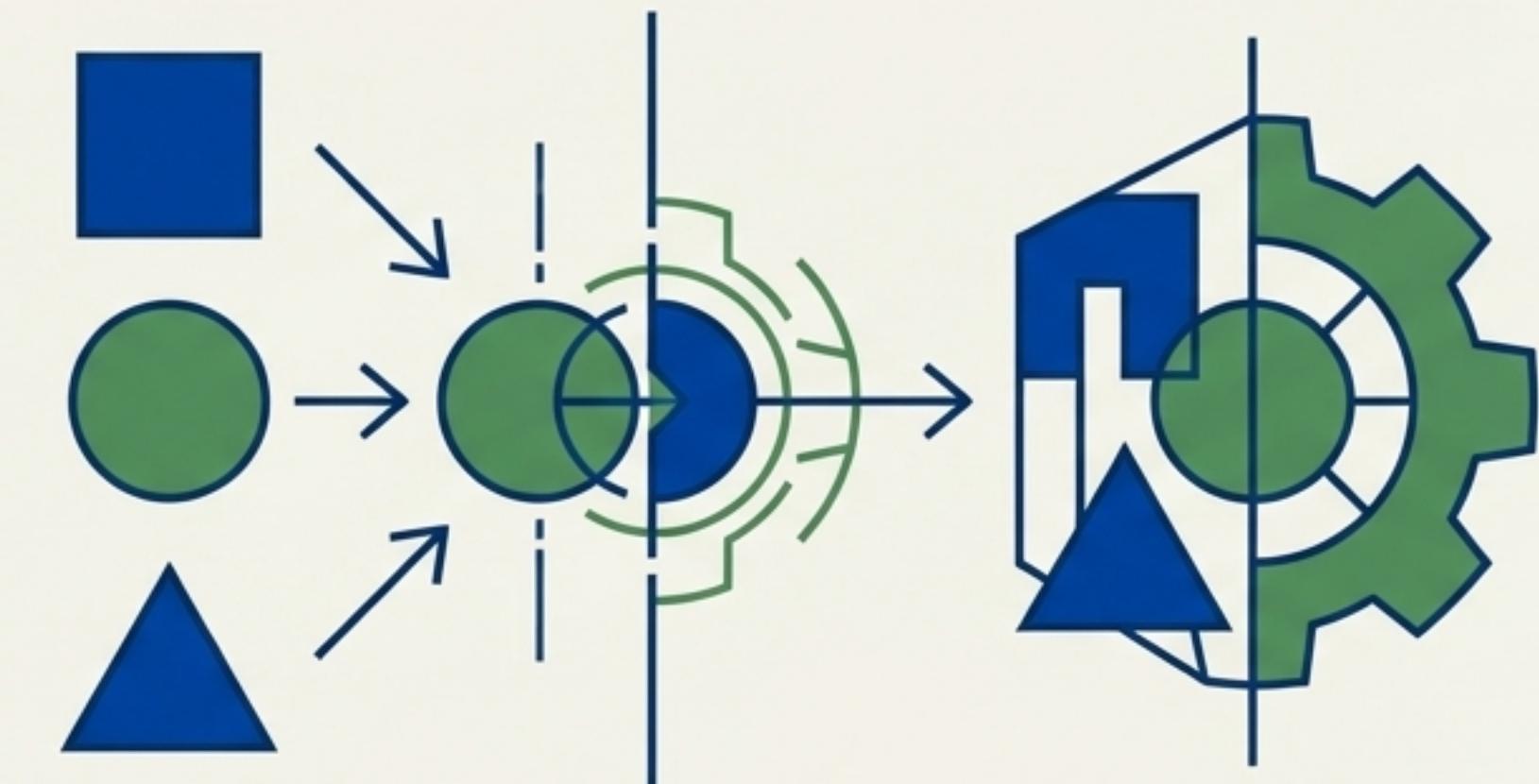
Задачи Линковщика, Шаг 2: Сборка и семантический анализ

Финальная сборка в единый виртуальный файл и проверка смысловой корректности кода.

5. Линковка: После всех проверок, содержимое всех файлов собирается в одно логическое дерево команд, готовое для интерпретатора.

6. Поиск семантических ошибок: Проверки, которые возможны до выполнения, но требуют знания контекста.

- Пример: Выход за пределы статически заданного массива. Если `game_field` определен как `[10, 10]`, а код пытается обратиться к ячейке `[11, 1]`, линковщик обнаружит эту ошибку.
- Динамические ошибки (например, выход за пределы массива с переменным размером) будут отловлены уже на этапе исполнения.



Философия ошибок: Максимальная информативность

В отличие от стандартного подхода Python, J-code стремится собрать и вывести все возможные ошибки за один проход.

```
ERROR: `game_rules.jcode: line 15` -  
Неизвестная команда 'move_sideways'.  
ERROR: `player_solution.jcode: line 42` -  
Неверный тип аргумента для команды 'for',  
ожидался словарь.  
ERROR: `player_solution.jcode: line 81` -  
Попытка доступа к статическому массиву за  
его пределами.
```

- **Цель:** Не останавливаться на первой проблеме, а предоставить программисту **полный список синтаксических и семантических ошибок**.
- **Механизм:** `CommandNode` хранит точный путь к файлу, строку и позицию символа для каждой команды. Это позволяет генерировать точные и полезные сообщения.
- **Типы ошибок:**
 - **Синтаксические** (роверяет линковщик)
 - **Семантические** (роверяет линковщик)
 - **Runtime-ошибки** (роверяет интерпретатор, например, бесконечная рекурсия).

От концепции к работающему интерпретатору

Проект находится в активной разработке, ключевые архитектурные решения приняты.

Что сделано

- ✓ Проработана философия и общая концепция.
- ✓ Определена архитектура, иерархия ролей и файловая структура.
- ✓ Разработан JSON-синтаксис и базовый класс `CommandNode`.
- ✓ Заложена основа линковщика и системы ошибок.

Что впереди

-  • Полноценная реализация всех этапов линковщика.
-  • Разработка интерпретатора, который будет исполнять дерево команд.
-  • Создание первой игры для демонстрации возможностей J-code.

Сложных вещей не бывает.

**«Сложных вещей не бывает,
бывают только интересные вещи,
которые достойны быть разбитыми
на подзадачи, бесстрастно и
бесстрашно быть решёнными».**

— Андрей Кубик



YouTube: Андрей Кубик